Energy–Efficient Instruction Dispatch Buffer Design for Superscalar Processors*

Gurhan Kucuk, Kanad Ghose,
Dmitry V. Ponomarev

Department of Computer Science
State University of New York
Binghamton, NY 13902–6000
ghose@cs.binghamton.edu

Peter M. Kogge

Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN 46556
kogge@cse.nd.edu

ABSTRACT

The instruction dispatch buffer (DB, also known as an issue queue) used in modern superscalar processors is a considerable source of energy dissipation. We consider design alternatives that result in significant reductions in the power dissipation of the DB (by as much as 60%) through the use of: (a) fast comparators that dissipate energy mainly on a tag match, (b) zero byte encoding of operands to imply the presence of bytes with all zeros and, (c) bitline segmentation. Our results are validated by the execution of SPEC 95 benchmarks on true hardware level, cycle–by–cycle simulator for a superscalar processor and SPICE measurements for actual layouts of the DB and its variants in a 0.5 micron CMOS process.

Keywords: Low–power superscalar datapath, low power comparator, low power instruction scheduling, bitline segmentation

1. INTRODUCTION AND BACKGROUND

Modern superscalar datapaths include a number of components for supporting out–of–order execution. In a K–way superscalar processor, instructions are fetched in program order and up to K instructions are *dispatched* to a dispatch buffer (DB, also called an *issue queue*) irrespective of the availability of the input operands. As results of prior instructions are computed, they are forwarded to waiting instructions in the DB. As soon as an instruction waiting in the DB has all of its input operands available, it becomes ready for execution. As soon as an execution unit ("function unit", FU) is available for a ready instruction, its operands are read out from the DB into the input latches of the selected FU – a process called *instruction issuing* – and execution commences.

As an instruction is dispatched, input registers that contain valid data are read out while the instruction is moved into the allocated DB entry. As the register values required as an input by instructions waiting in the DB (and in the dispatch stage) are produced, they are forwarded through forwarding buses that run across the length of the DB [7]. To balance the overall flow in the pipeline, at least K sets of forwarding paths are provided to allow K different results from FUs to be forwarded to instructions waiting in the DB. We consider a datapath where the DB entry for an instruction has one data field for each input operand, as well as an associated tag field that holds the address of the register whose value is required to fill the data field. When a function unit completes, it puts out the result produced, along with the address of the destination register for this result on a forwarding bus.

Comparators associated with invalid operand slots of valid DB entries then match the tag values stored in the fields (for waited–on register values) against the destination register address floated on the forwarding bus [7]. On a tag match, the result floated on the bus is latched into the associated input field. Since multiple function units complete in a cycle, multiple forwarding buses are used; each input operand field within a DB entry thus uses a comparator for each forwarding bus. Examples of processors using this datapath style are the Intel Pentium Pro, Pentium II, IBM Power PC 604, 620 and the HAL SPARC 64.

Figure 1 depicts the black box view of a DB as described above. This is essentially a multi–ported register file with additional logic for associative data forwarding from the forwarding buses and associative addressing logic that locates free entries and entries ready for issue. We assume a 4–way superscalar processor for our studies. The DB is assumed to have 4 read ports, 4 writes ports and 4 forwarding buses. The 4 write ports are used to establish the entry for up to 4 instructions simultaneously in the DB at the time of dispatching. The four read ports are used to select up to 4 ready instructions for issue and move them out of the DB to the FUs. The main sources of energy dissipation in the DB are as follows:

a) Energy dissipated in the DB in the process of establishing DB entries for dispatched instructions: in locating a free entry associatively and in writing into the selected entry.

b) Energy dissipated in the DB when FUs complete and forward the results and/or status information to the DB entries. A significant fraction of this energy dissipation is due to the tag comparators used for associative matching to pick up forwarded data.

c) Dissipations in the DB at the time of issuing instructions to the FUs: in arbitrating for the FUs, enabling winning instructions for issue and in reading the selected instructions and their operands from the DB.

The energy dissipation within the DB is a significant power dissipation component for modern superscalar CPUs. In this paper we examine the use of several techniques for reducing these dissipations without compromising the cycle time of the CPU. For this purpose, power measures are obtained using detailed cycle–by–cycle, true hardware level simulations of the SPEC 95 benchmarks and the use of SPICE measurements of actual 0.5 micron layouts of the DB – an approach that is as good as it gets short of an implementation. Our goal was to maintain a 3.3 nS cycle time for the processor despite the proposed microarchitectural and circuit changes for reducing the DB power. This cycle time was dictated by the delays

ISLPED '01, August 6–7, 2001, Huntington Beach, California, USA.
Copyright 2001 ACM 1–58113–371–5/01/0008...$5.00.

* supported in part by DARPA through contract number FC 306020020525 under the PAC–C program, by the IEEC at SUNY–Binghamton & the NSF through award nos. MIP 9504767 and EIA 9911099.

of cache layouts in the same technology, as used in some earlier studies. Irrespective of the technology used, we believe our approach is fairly universal in reducing dynamic dissipations in the DB.

In an earlier paper [5], we examined a technique for splitting up a DB into distributed DBs tailored for specific instruction types and the suppression of leading zeros in operands to save power. As wire delays become significant, such a distribution may impact the cycle time adversely. We focus this paper on techniques that avoid such a distribution and maintain the centralized structure of the DB, with minimal impact on the circuit design and layouts. We examine the impact of three main techniques, used singly or in conjunction, in this respect. The techniques used are also applicable to other superscalar datapath structures like the reorder buffer as well as multiported register files; the relevant results for these other structures are not reported here.

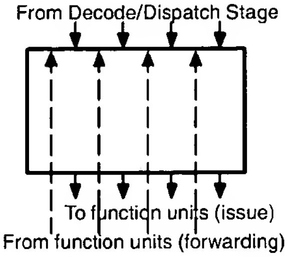

From Decode/Dispatch Stage
To function units (issue)
From function units (forwarding)

Figure 1. Black box view of the Dispatch Buffer

2. REDUCING DISPATCH BUFFER POWER: 3 APPROACHES

We examine the use of three relatively independent techniques for reducing the DB power dissipations. To set the right context, it is useful to examine the major power dissipation components within the DB. Figure 2 shows these components, measured using our technique, averaged over the SPEC 95 integer and floating point benchmarks for the base case DB, which is a traditional design using comparators that dissipate energy on a tag mismatch. For both integer and floating point benchmarks, issue power is the dominant component. For floating point benchmarks, data forwarding contribution to the total power dissipation is relatively higher than for integer benchmarks, because of the higher latency of floating point operations which increases the average number of DB operand slots waiting for the results. Simulation of SPEC 95 benchmarks within our experimental framework shows that on the average about 21 slots are waiting for the results during the execution of integer benchmarks and 25 slots are waiting during the execution of floating point benchmarks. All such slots employ the traditional pulldown comparators (that dissipate energy on tag mismatches) in current implementations of DBs [7].

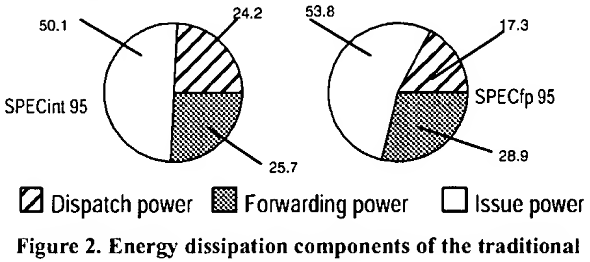


Figure 2. Energy dissipation components of the traditional DB (% of total)

We thus explore the use of fast comparators for tag matching that dissipate energy mainly on a tag match as the technique that directly reduces the energy spent in data forwarding. We then explore the use of zero byte encoding [1, 4, 5, 8] to reduce the number of bit lines activated during dispatching, forwarding and issuing. Finally, we add bit-line segmentation to reduce energy dissipations in the bit lines that are driven. The overall energy saving realized for the DB by using all of these techniques in combination is about 60% and this is realized without any increase in the cycle time of the processor but with an acceptable increase in the silicon real estate for the DB.

2.1 Using Energy-Efficient Comparators in the DB

The typical comparator circuitry used for these associative matching in a DB is a dynamic pulldown comparator or a 8-transistor associative bitcell. Such comparators have a fast response time to allow matching and the resulting updates to be completed within the cycle time of the processor. All of these comparators dissipate energy on a *mismatch* in any bit position. A significant amount of energy is thus wasted in comparisons that do not locate matching entries, while little energy (in the form of precharging) is spent in locating matching entries. *Typically, comparisons are enabled only for the DB entry slots that are waiting for a result*; in a cycle only a small percentage of these slots actually match the destination address of the forwarded data. As an example, the data collected for the simulated execution of SPEC 95 benchmarks on our system indicate that about 23 operand slots out of the 128 that we have in our 64-entry dispatch buffer are actually waiting for results (i.e., the comparators for these slot are enabled for comparison). Out of these, only 2 to 4 comparators produce a match per cycle on the average. This is clearly an energy-inefficient situation, as more energy is dissipated due to mismatches (compared to the number of matches) with the use of traditional comparators that dissipate energy on a mismatch. Similar observations are valid for re-order buffers which double as physical registers (as used in the X86 implementations, for example), where associative matching is used to locate the most recently established entries for instruction destinations. We propose to remedy this by designing and using fast comparators that (predominantly) dissipate energy on a match.

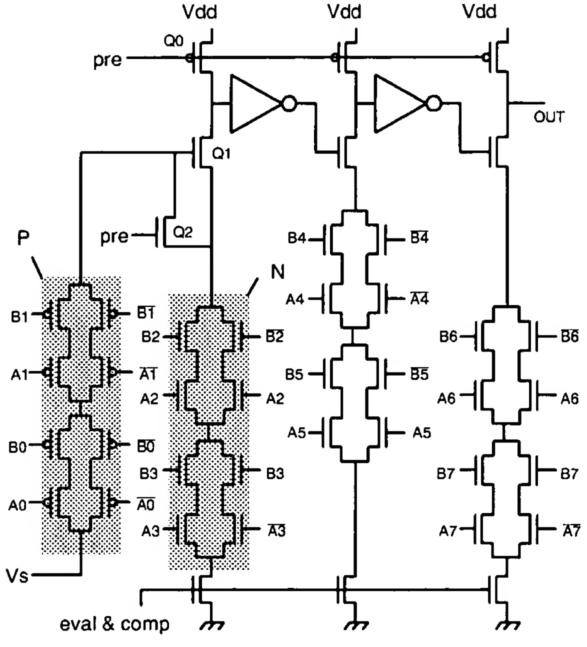


Figure 3. The proposed comparator

In theory, one can design CMOS comparators to dissipate energy dynamically only on a full match but these designs require a large number of transistors and/or switch slowly. Instead of choosing such a design, we opted for a comparator design that dissipates minimum energy on (infrequent) partial mismatches in the comparand values and dissipates an acceptable amount of energy on a full match. This comparator consciously takes into account the characteristics of the distribution of the physical register addresses that are compared in a typical dispatch buffer to minimize energy dissipations for partial matches.

Table 1 shows how comparand values are distributed and the extent of partial matches in the values of two adjacent bits in the comparands, averaged over the simulated execution of SPEC 95 integer and floating point benchmarks, as well as the average over both the integer and floating point benchmarks. The lower order 4 bits of both comparands are equal in roughly 10.3% of the case, while the lower order 6 bits match 5.54% of the time. A full match occurs 4.93% of the time on the average. Equivalently, a mismatch occurs in the lower order 4 bits of the comparator 89.7% of the time (=100 – 10.3). The behavior depicted in Table 1 is a consequence of the localization of dependencies in typical source code. This causes physical registers from localized regions to be allocated as the destination of instructions involved in the dependency (The addresses of these physical registers are compared within the DB to satisfy data dependencies.) Consequently, the likelihood of a match of the higher order bits of the register addresses (i.e., the comparands) is higher. Our comparator design directly exploits this fact by limiting dynamic energy dissipation due to partial matches to less than 10.3% of cases when the lower order 4 and 6 bits match; no energy dissipation occurs in the more frequent cases of the higher order bits matching. The overall energy dissipation due to partial mismatches is thus greatly reduced. Energy dissipation occurs, of course, on a full match but this dissipation is smaller than that of a comparable traditional comparators (that pull down a precharged line only on a mismatch in one or more bit positions.)

Figure 3 depicts our proposed comparator for comparing two 8–bit comparands, A7A6..A0 and B7B6..B0, where 0 is the least significant bit. The basic circuit is a three stage domino logic, where the first stage detects a match of the lower 4 bits of the comparands. The following two stages do not dissipate any energy when the lower order 4 bits do not match. The precharging signal is cut off during the evaluation phase (when eval is active) and an evaluation signal is applied to each stage of the domino logic only when the comparison is enabled (comp is high). The series structure P composed of 8 p–transistors passes on a high voltage level (Vs, where Vs is < Vdd but higher than lower threshold for a logic 1 level) to the gate of the n–transistor Q1 only when the two lower order bits of the comparands match. The series structure N turns on when the next pair of lower order bits of the comparands (A3A2 and B3B2) match. When comparison is enabled, the output of the first stage (driving an inverter) goes low during the evaluation phase only when all lower order 4 bits of the comparands match. Till such a match occurs, no dynamic energy is dissipated in the other stages of the comparator. Transistor Q2 is needed to prevent Q1 from turning on due to the presence of charge left over on its gate from a prior partial match of the two lower order bits. The charge moved from the gate of Q1 by Q2 is dissipated to ground only when there is a subsequent match in bits 3 and 2 (which turns the structure N on.) This effectively reduces dissipations in the case when only the two lower order bits match; this dissipation is further minimized by keeping Vs slightly lower than Vdd. As in any series structure of n–transistors that pull down a precharged line, the W/L ratios of the n–devices go up progressively from the top to the bottom (ground). The p–transistors in the structure P, the precharging transistors and the inverters are sized to get an acceptable response time on a match. This comparator can respond quickly enough to let the match proceed and allow the matching data be latched into the appropriate field of the DB entry within the targeted cycle time of 3.3 nsec. for our 0.5 micron layouts.

Number of bits matching —>	% of total cases			
	2 LSBs	4 LSBs	6 LSBs	All 8 bits
Avg, SPECint 95	27.2	9.8	5.7	5.6
Avg, SPECfp 95	33.1	10.7	5.4	4.4
Avg, all SPEC 95	30.5	10.3	5.6	4.9

LSB = least significant bits

Table 1. Dispatch Buffer Comparator Statistics

The comparator of Figure 3 actually has a lower dissipation on a match and faster response time compared to traditional parallel pulldown comparators that discharge a precharged line on a mismatch in any bit position. This is because the effective output loading of traditional (mismatch) comparators is high, amounting to the diffusion capacitances of 2*C n–transistors (C is the number of bits compared = 8).

2.2 Using Zero Byte Encoding

A study of data streams within superscalar datapaths as reported in [6] shows that significant number of bytes are all zeros within operands on most of the flow paths (dispatch stage to DB, DB to function units, functions units to destinations and forwarding buses etc.). On the average, in the course of simulated execution of the SPEC 95 benchmarks on cycle accurate and true register level simulators, about half of the byte fields within operands are all zeros. This is really a consequence of using small literal values, either as operands or as address offsets, byte–level operations, operations that use masks to isolate bits etc. Considerable energy savings are possible when bytes containing zero are not transferred, stored or operated on explicitly. Other work in the past for caches [9], function units [1] and *scalar* pipelines have made the same observation. We extend these past work to superscalar datapaths, where additional datapath artifacts to support out–of–order execution can benefit from the presence of zero–valued bytes. The DB is an example of just such an artifact.

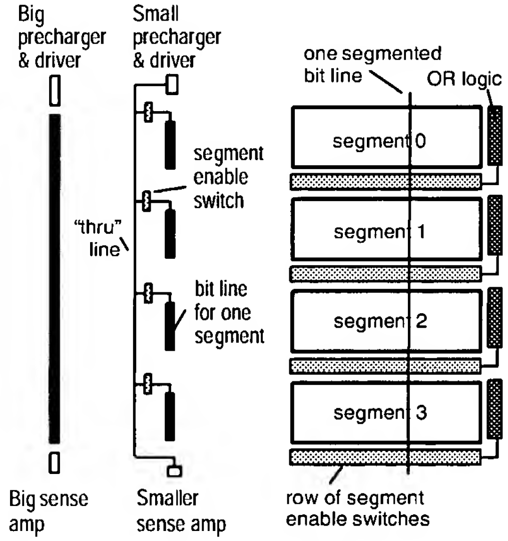
By not writing zero bytes into the DB at the time of dispatch, energy savings result as fewer bitlines need to be driven. Similarly, further savings are achieved during issue by not reading out implied zero valued bytes. This can be done by storing an additional bit with each byte that indicates if the associated byte contains all zeros or not. The contents of this zero indicator bit can be used to disable the word select strobe from going to the gates of the pass transistors. By controlling the sensitivity of the sense amps, we can also ensure that sense amp transitions are not made when the voltage difference on differential bit lines is below a threshold (as would be the case for the precharged bitline pairs associated with the bitcells whose readouts are disabled as described above. Relevant circuit details are beyond the scope of this paper; some circuitry is described in [6]. Zero–valued bytes do not have to be driven on the forwarding buses that run through the DB – this is another source of energy savings that result from zero byte encoding.

The price paid for energy savings within the DB through the use of zero byte encoding is in the form of an increase in the area of the DB by about 11%. There is a very slight increase in the DB access times, but it still allows the target cycle time of 3.3 nS for the entire datapath to be maintained.

2.3 Using Bitline Segmentation in the DB

As mentioned earlier, the DB is essentially a register file with additional associative logic for data forwarding. The DB is written to using the normal logic for a register file at the time of dispatch, to set up the DB entry for dispatched instructions. For each instruction dispatched in a cycle, a write port is needed. The only difference from a normal register file is that the word being written to from each write port is selected associatively (an associative search is needed to locate free entries, i.e., words within the DB), instead of being selected through an explicit address. At the time of instruction issue, instructions ready for execution are read out from the DB through independent read ports. Other than the use of the wakeup and arbitration logic to select ready entries, this readout is identical to what happens when a normal register file is read out. Sense amps, similar to what are used in RAMs are needed to sense the data being read out as the bit lines are fairly long. As in a multiported RAM or register file, the bit lines in the DB are a significant source of energy dissipation in the course of instruction dispatching (writes) and instruction issuing (reads). The bitlines associated with each read and write port present a high capacitive load, which consists of a component that varies linearly with the number of rows in the DB. This component is due to the wire capacitance of the bitlines and the diffusion capacitance of the pass transistors that connect bitcells to the bit lines.

The capacitive loading presented by the bitlines in the DB can be reduced through the use of bitline segmentation. The entire DB is viewed as a linear array of segments for this purpose, with consecutive bitcell rows making up a segment. Figure 4 (a) is useful in understanding how bitline segmentation reduces the capacitive loading encountered during reads and writes from the bit line. As an example, consider a DB with 64 rows which has been restructured into 4 segments. Each segment will then consist of 16 consecutive rows. The original bit line, shown in the left of Figure 4 (a) is loaded by the diffusion capacitance of 64 pass devices, the diffusion capacitances of the precharging and equibrator devices, sense amp input gate capacitances and the diffusion capacitances of tri-stated devices used to drive the bit lines (during a write). In addition, there is the wire capacitances of the bit line itself. In the segmented version, the bit line is split into four segments; each segment of the bit line covers a column of the bitcells of the rows within a segment. As a result, the capacitive loading on each segment is lowered: each segment is connected to only 16 pass devices and the wire length of the bit line segment is one fourth of the original bit line.



(a) Original (left) and segmented (right) bit line

(b) Segmented DB

Figure 4. Bitline segmented DB

To read a bitcell within a segment, the bitline for that segment has to be connected to the precharger and sense amp; for a write, the bitline segment has to be connected to the tri-state enable devices for the bit line driver. This is accomplished by running a "through" wire across all of the segments (typically in a different metal layer, right over the segmented bit lines), which is connected to the prechargers, sense amp and tri-state drivers as in the non-segmented design. A switch is then used to connect the segment in question to this "through" line. For the DB, the segment switch is turned on by OR-ing the associative read or write enable flags for the port (associated with the bitline) for all the rows in that segment. As the effective capacitive loading on the through line and the connected bitline segment is smaller than the capacitive loading of the unsegmented bit line, lower energy dissipation occurs during reads or writes. The savings are further enhanced due to the use of smaller devices for precharging, sensing and driving. On the down side, additional energy dissipation occurs in the logic associated with the control logic for the segment enable switch, the energy needed to drives the switches for all of the columns and the loading imposed on the through line by the diffusion capacitances of the complementary segment enabling switches. By carefully choosing the size of a segment, the overall energy dissipations can be minimized – making the size of a segment too small can actually increase the overall energy consumption, while making the size too large defeats the purpose of segmentation. An optimal segment size of 8 rows was discovered for the 64-entry DB described here.

Figure 4(b) depicts a segmented DB and shows that there is some increase in the overall area of the DB due to the use of a row of segment enable switches with each segment. For the 0.5 micron CMOS layouts used here, the overall growth of the layout area of the 64-entry DB when it was segmented into 8 segments (8 rows per segment) was only about 5%.

3. EVALUATION METHODOLOGY AND RESULTS

We rely on true hardware level cycle by cycle simulations to measure the actual number of energy dissipating transitions in various parts of the datapath. The well-known Simplescalar simulator [2] was extensively modified (only 20% of the original source code is retained!) for this purpose and the execution of the SPEC 95 benchmarks were simulated (each benchmark was run for 200 million instructions after a 200 million instructions startup phase). Detailed accounting for detecting zero bytes in operands and lower level transition counting was implemented by a separate thread. Transition counts for reads, writes, associative addressing, FU arbitration, tag matching, data latching and other notable events were recorded separately. Transition counts and other data gleaned from the simulator were then fed into a power estimation program that used dissipation energies measured using SPICE for actual 0.5 micron layouts of key datapath components. (The process used was a 0.5 micron 4 metal layer CMOS process, HPCMOS-14TB; we are in the process of migrating our designs to 0.18 micron process.) Our power estimation program generated power measures in milliwatts for major energy dissipating events within the DB for each benchmark in the SPEC 95 suite individually as well as for the averages of the integer and floating point benchmarks and total averages.

The configuration of the system studied was as follows. The L1 I-cache and L1 D-cache were both 32 KBytes in capacity with a line size of 32 Bytes, with the former being direct-mapped and the latter being 4-way set-associative. A 4-way set-associative, integrated L2 cache with a capacity of 512 KBytes and a line size of 64 Bytes was assumed. The size of the dispatch buffer and the re-order buffer were kept at 64 entries and 100 entries respectively. The physical register file for integers and floats were 128 in number each. The function units are as follows: 4 integer units, one integer multiply/divide unit, 4 floating point multiply-add units, one floating point multiply/divide unit one load unit and one store unit. The latencies were as used in the original Simplescalar simulator. A 4-way dispatch, a and a 4-way commitment were assumed. We assumed that when a function unit produces a 32-bit result, it will drive at most 32 bits even if wider connections are available. We assume this to be true for the base cases as well to make our comparisons fair.

Figure 5 (a) shows the effects of the new comparator on the power dissipated during the process of instruction forwarding and tag matching. Energy reduction is as high as 45% in this case. However, the total DB power is reduced by only about 12% because the comparator does not have any effect on the energy dissipated during instruction dispatch and issue (Figure 5 (b)). Zero-byte encoding and bitline segmentation address the power reduction in exactly these components. Figures 5 (c) and 5 (d) show the power dissipated during the dispatch and issue respectively and the corresponding savings achieved by zero-byte encoding, bitline segmentation and the combination of these two techniques. During dispatch, bitline segmentation leads to about 53% of power savings, zero-byte encoding results in about 23% reduction, and combined savings are more than 60%. At issue, segmentation savings are 41%, zero-byte encoding savings are 35% and the combined power reduction is 60%. In addition, zero-byte encoding achieves extra 13% reduction in power dissipation during forwarding on top of what is realized by deploying the new comparator. Figure 5 (e) summarizes the results and shows the total power dissipation within the dispatch buffer and energy savings realizable using some combinations of the aforementioned mechanisms. Bitline segmentation is a powerful technique on its own, resulting in about 32% energy reduction in the DB. Savings attributed to the use of zero-byte encoding are about 26% (this is not shown in the graph). Segmentation and zero-byte encoding in concert reduce the energy by more than 46%. Notice that the total savings achieved by the two techniques is not the sum of their individual respective savings. This is because bitline segmentation reduces the lengths of the bitlines which somewhat reduces the savings achieved by zero-byte encoding. The combination of all three techniques – bitline segmentation, zero-byte encoding and the use of fast, power-efficient comparators achieve a remarkable 60% reduction in power dissipated by the instruction dispatch buffer of a superscalar CPU. Since zero-byte encoding reduces the power dissipation during forwarding by about 15%, the effects of the new comparator on power savings are a little smaller than 14% (reported above) if the new comparator is applied to segmented DB with zero-byte encoding.

Figure 6 shows the results for a different datapath configuration where register operands are read out at the time of instruction issue. In this case, the DB entry has status bits for each input register in an instruction; register operand values are not part of the DB entry. Here forwarding simply amounts to updating the status of matching input physical registers within established DB entries. When all input registers of a DB entry are ready, the corresponding instruction is ready for issue and operands are read out from the physicl registers (or bypassed to the FU inputs) as the instruction issues. As seen from Figure 6, power savings achieved with the use of the new comparator are relatively higher for this datapath variant. This is expected, as the comparator dissipations dominate the power expended in forwarding since no actual data values are forwarded to the DB entries. Savings in power dissipation during forwarding is 70% on the average with the use of the new comparator (Figure 6 (a)). The overall power savings within the DB with the use of the new comparator average about 25% (Figure (b)). Overall, the average power savings for this datapath variant are: (i) 22% with just the use of segmentation, (ii) 25% with only the use of zero byte encoding; (iii) 38% with the use of segmentation and zero byte encoding and (iv) 63% with the use of the new comparators, zero byte encoding and segmentation.

Note that in all of the power savings computations relative to the base case, we assumed that forwarding comparisons are enabled only for DB entries that are awaiting a result; comparisons are not done for unallocated entries or allocated entries that have already been forwarded the result or the status of the result (the latter for the datapath variation discussed). Any sensible design – and ones that we are aware of – should do this anyway. If spurious comparisons are allowed, the power savings reported here will go up further.

Finally, note also that segmentation is a form of selective dynamic activation of DB entries, along the lines of dynamic allocation of DB partitions, as done in [3] and for the DB and ROB in [8]. The new comparator and zero byte encoding technique can be used in conjunction with dynamic resource allocation of datapath resources, as in [3, 8]. These extensions are the subject of a forthcoming paper.

4. CONCLUSIONS

We studied three relatively independent techniques to reduce the energy dissipation in the instruction dispatch buffers of modern superscalar processors. First, we proposed the use of fast comparators in forwarding/tag matching logic that dissipate the energy mainly on the tag matches. Second, we considered the use of zero-byte encoding to reduce the number of bitlines that have to be driven during instruction dispatch and issue as well as during forwarding of the results to the waiting instructions in the DB. Third, we evaluated power reduction achieved by the segmentation of the bitlines within the DB. Combined, these three mechanisms reduce the power dissipated by the instruction dispatch buffer in superscalar processors by more than 60% on the average across all SPEC 95 benchmarks.

The DB power reductions are achieved without compromising the cycle time and only through a modest growth in the area of the DB (about 12%, including the new comparators, ZE logic and segmentation). Our ongoing studies also show that the use of all of the techniques that reduce the DB power can also be used to achieve reductions of a similar scale in other datapath artifacts that use associative addressing (such as the reorder buffer and load/store queues). As the power dissipated in instruction dispatching, issuing, forwarding and retirement can often be as much as half of the total chip power dissipation, the use of the new comparators, zero byte encoding and segmentation offers substantial promise in reducing the overall power requirements of contemporary superscalar processors.

5. REFERENCES

[1] Brooks, D. and Martonosi, M., "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", in Proc. HPCA, 1999.

[2] Burger, D., and Austin, T. M., "The SimpleScalar Tool Set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin, Madison, June 1997 and documentation for all Simplescalar releases (through version 3.0).

[3] Buyuktosunoglu, A. et al, "An Adaptive Issue Queue for Reduced Power at High Performance", Proc. PACS workshop, held in conjunction with the 9-th ASPLOS, 2000.

[4] Canal, R., Gonzales, A., and Smith, J., "Very Low Power Pipelines using Significance Compression", in Proc. Micro33, 2000.

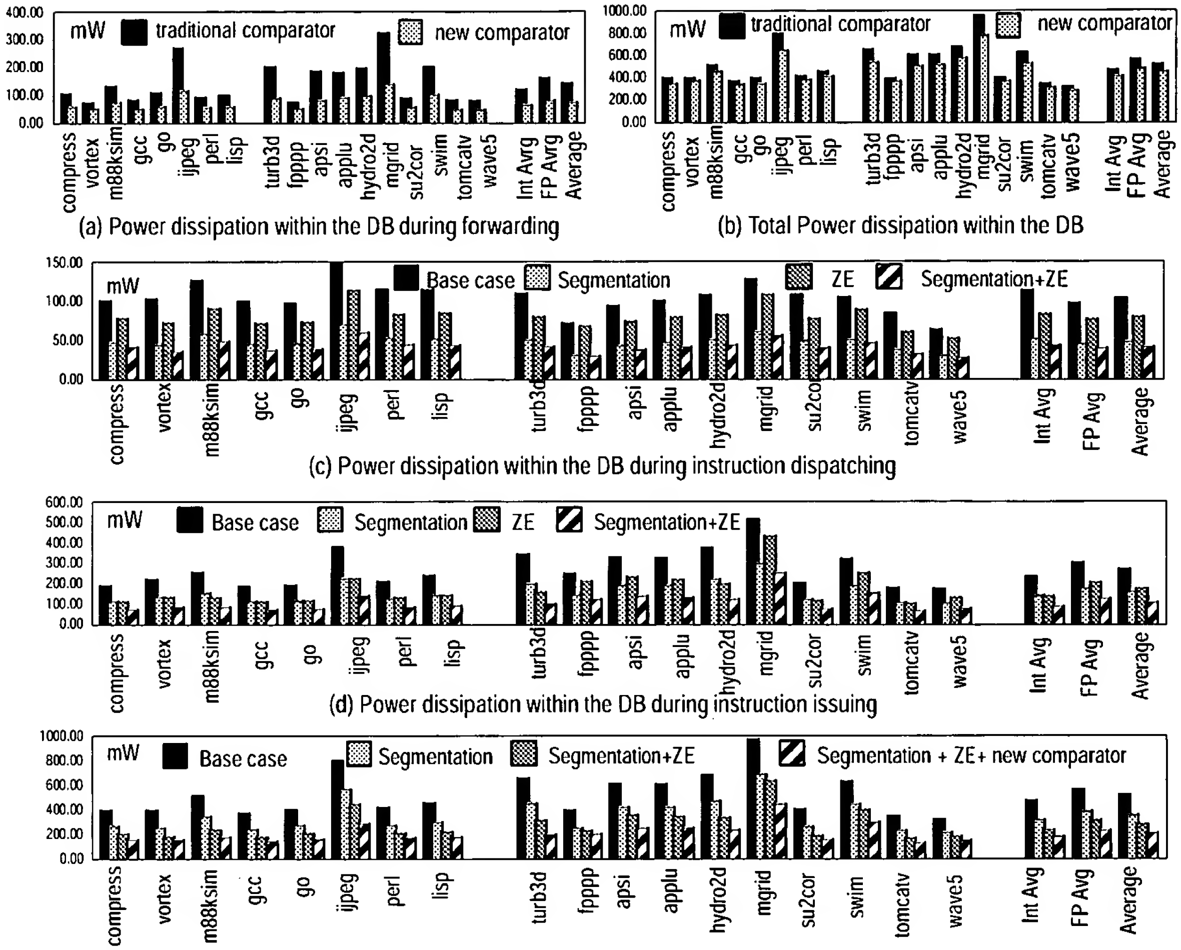
[5] Ghose, K., "Reducing Energy Requirements for Instruction Issue and Dispatch in Superscalar Microprocessors," Proc. ISLPED, 2000, July 2000, pp.231–234.

[6] Ghose, K., Ponomarev, D., Kucuk, G., Flinders, A., Kogge, P., and Toomarian N.,"Exploiting Bit-slice Inactivities for Reducing Energy Requirements of Superscalar Processors," in Proc. of Kool Chips Workshop, Micro-33, 2000.

[7] Palacharla, S., Jouppi, N. P. and Smith, J. E., "Quantifying the complexity of superscalar processors", Tech. Report 96-1308, Dept. of CS, Univ. of Wisconsin, Madison, 1996.

[8] Ponomarev, D., Kucuk, G. and Ghose, K., "Dynamic Allocation of Datapath Resources for Low Power", to appear in the Proc. of the Workshop on Complexity-Effective Architectures, held in conjunction with ISCA, June 01.

[9] Villa, L., Zhang, M. and Asanovic, K., "Dynamic Zero Compression for Cache Energy Reduction", in Proc. of Micro-33, 2000.



(a) Power dissipation within the DB during forwarding

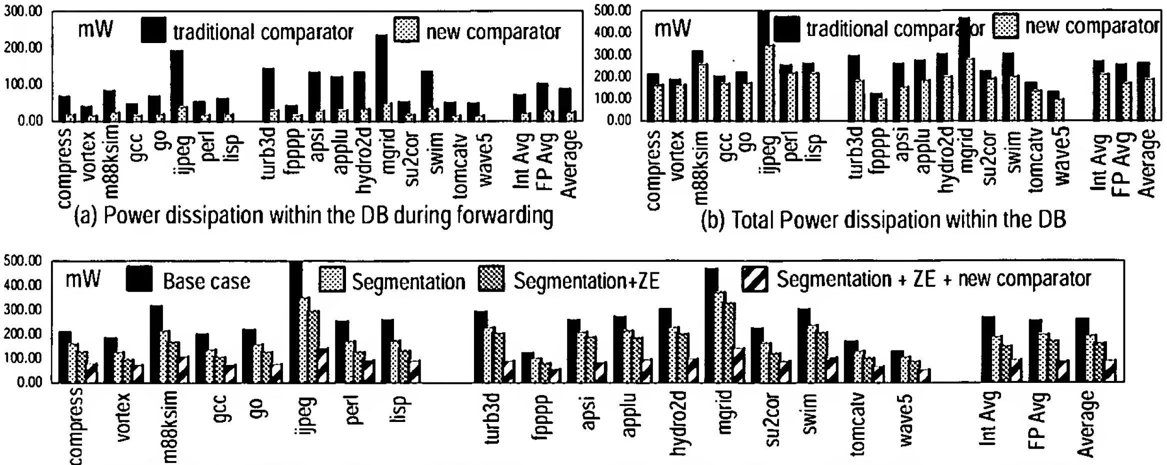
(b) Total Power dissipation within the DB

(c) Power dissipation within the DB during instruction dispatching

(d) Power dissipation within the DB during instruction issuing

(e) Total power dissipation within DB using bitline segmentation, power efficient comparators and zero encoding

Figure 5. Power Savings Achieved in the Dispatch Buffer



(a) Power dissipation within the DB during forwarding

(b) Total Power dissipation within the DB

(c) Total power dissipation within DB using bitline segmentation, power efficient comparators and zero encoding

Figure 6. Power Savings Achieved in the Dispatch Buffer for Datapath Variant Where All Register Operands are Read Out at the Time of Instruction Issue